String Matching

Algorithm : Design & Analysis

[18]

In the last class…

- Optimal Binary Search Tree
- Separating Sequence of Word
- Dynamic Programming Algorithms

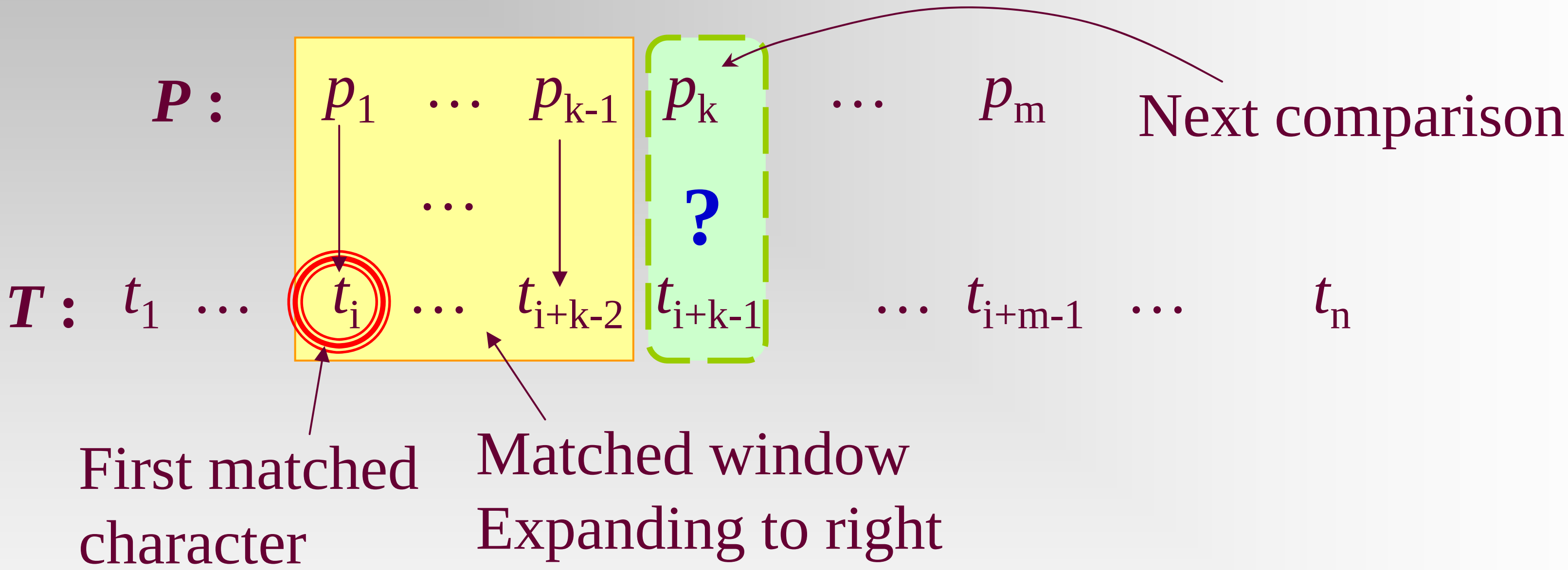
String Matching

- Simple String Matching
- KMP Flowchart Construction
- Jump at Fail
- KMP Scan

String Matching: Problem Description

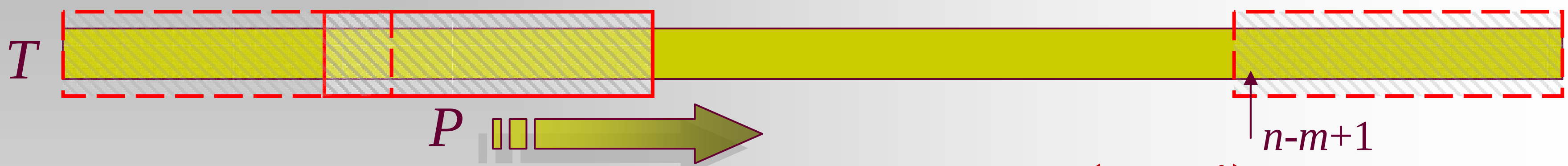
- **Search the text** *T*, a string of characters of length *n*
- **For the pattern** *P*, a string of characters of length *m* **(usually, m<<n)**
- The result
  - If *T* contains *P* as a substring, returning the index starting the substring in *T*
  - Otherwise: fail

Straightforward Solution



Note: If it fails to match p_k to t_{i+k-1}, then backtracking occurs, a cycle of new matching of characters starts from t_{i+1}.In the worst case, nearly n backtracking occurs and there are nearly m-1 comparisons in one cycle, so **$\Theta(mn)$**

Brute-Force, Not So Bad as It Looks



sliding window **worst-case: *m*(*n*-*m*+1)**

Average-case: (characters of *P* and *T* randomly chosen from Σ ($|\Sigma|$=d≥2)

For a specific window, the expected number of comparison is:

matched: $m\left(\frac{1}{d}\right)^m$

ummatched: for the case that the first unmatched character

is the ith in the window, then, $i\left(\frac{1}{d}\right)^{i-1}\left(1-\frac{1}{d}\right)$

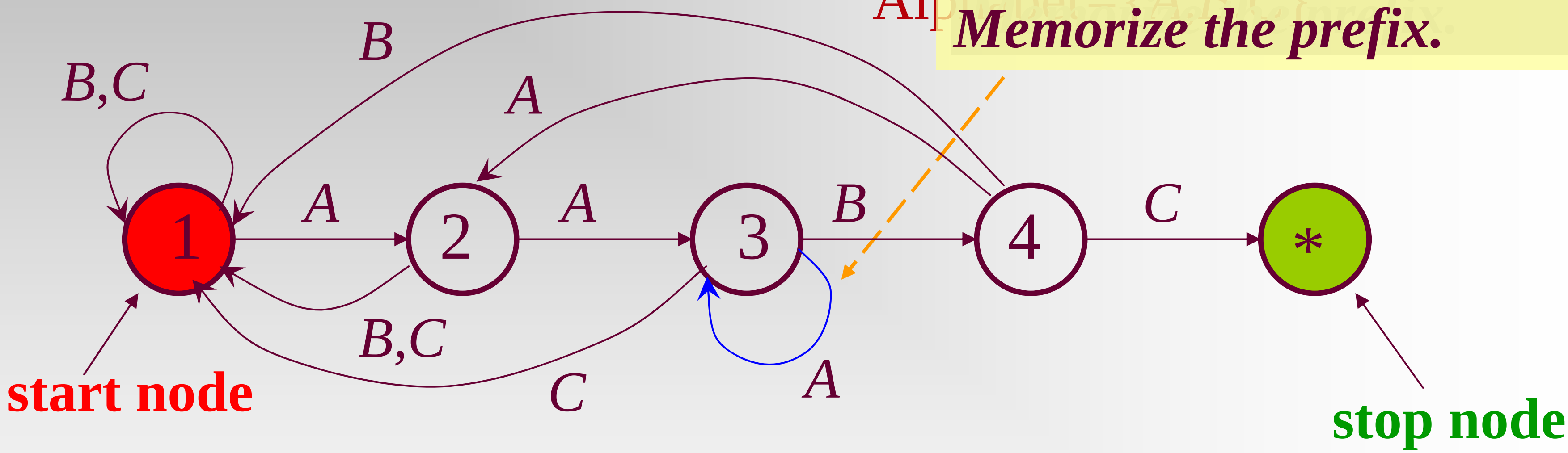
So, $\sum_{i=1}^{m}\left[i\left(\frac{1}{d}\right)^{i-1}\left(1-\frac{1}{d}\right)\right]+m\left(\frac{1}{d}\right)^m=1+\sum_{i=1}^{m}\left[(i+1)\left(\frac{1}{d}\right)^i-i\left(\frac{1}{d}\right)^i\right]=\frac{1-d^{-m}}{1-d^{-1}}\le 2$

Disadvantages of Backtracking

- More comparisons are needed
- Up to *m*-1 most recently matched characters have to be readily available for re-examination. (Considering those text which are too long to be loaded in entirety)

An Intuitive Finite Automaton for Matching a Given Pattern

Why no backtracking?
Memorize the prefix.

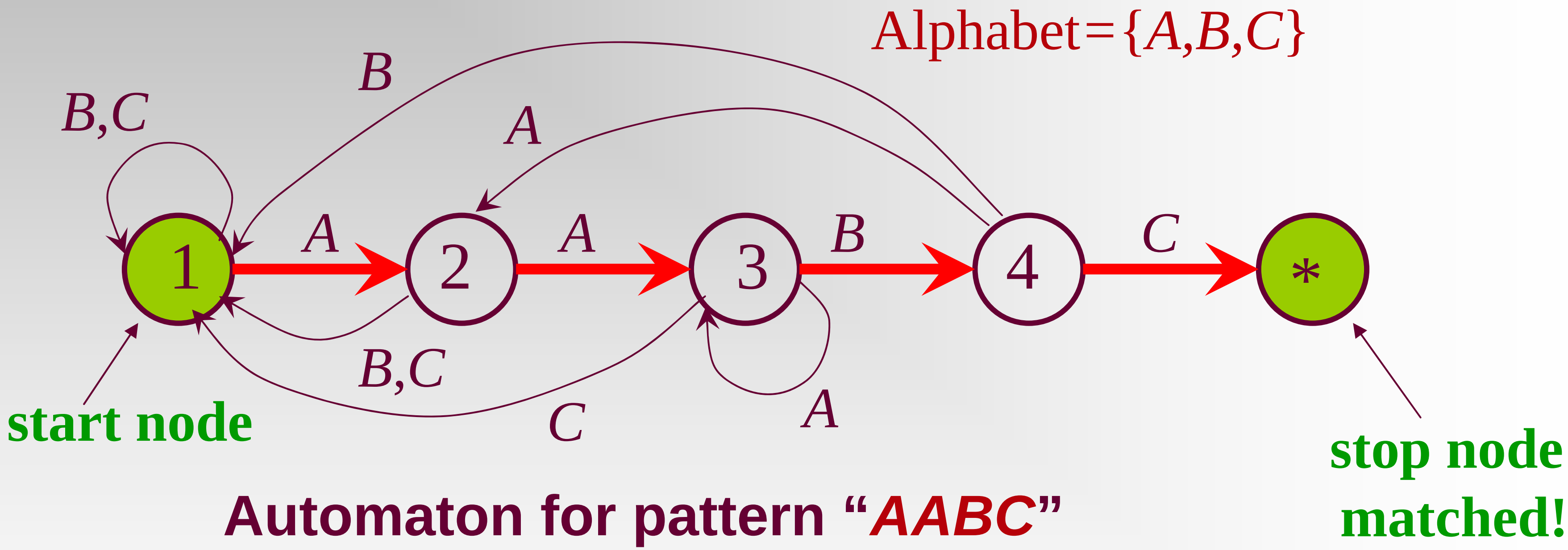


Automaton for pattern "*AABC*"

Advantage: each character in the text is checked only once

Difficulty: Construction of the automaton – too many edges(for a large alphabet) to defined and stored

Looking at the Automata Again

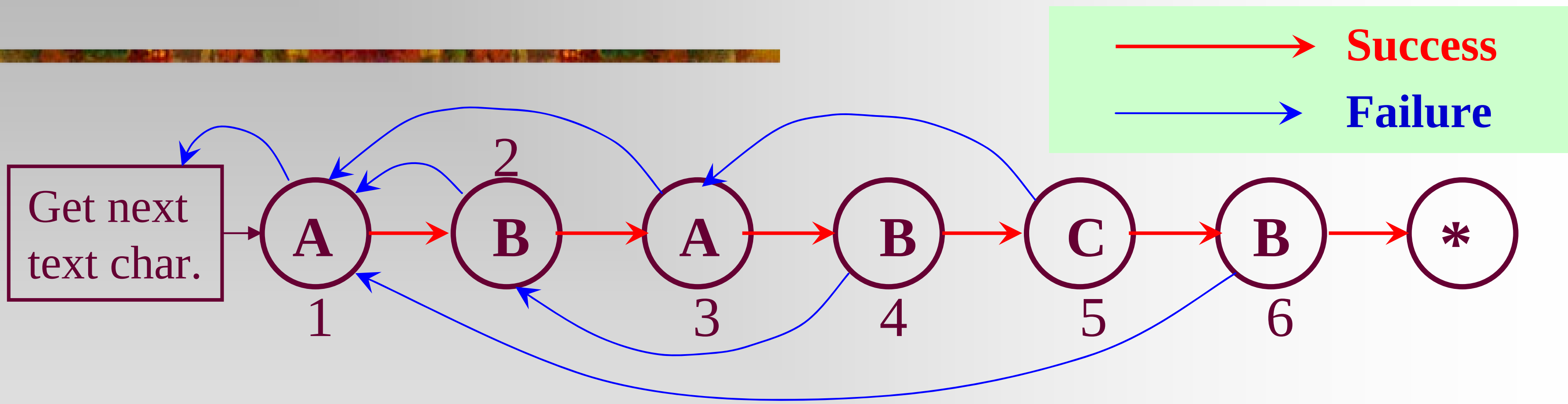


Automaton for pattern "*AABC*"

There is only one path to success,

However, many paths leading to Fail.

The Knuth-Morris-Pratt Flowchart



1 2 3 4 5 6 7 8 9 10 11

An example: *T*="A C A B A A B A B A", *P*="ABABCB"

KMP cell number	1	2	1	0	1	2	3	4	2	1	2	3	4	5	3	4
Text being scanned	1	2	2	2	3	4	5	6	6	6	7	8	9	10	10	11
	A	C	C	C	A	B	A	A	A	A	B	A	B	A	A	-
Success or Failure	s	f	f	C	s	s	s	f	f	s	s	s	s	f	s	**F**

get next char.

Matched Frame

P: ABABABCB

T: … ABABAB *x* …

to be compared next

matched frame

If *x* is not C

P: ABAB ABCB

T: … ABABAB *x* …

The matched frame move to right for 2 chars, which is equal to moving the pointers backward.

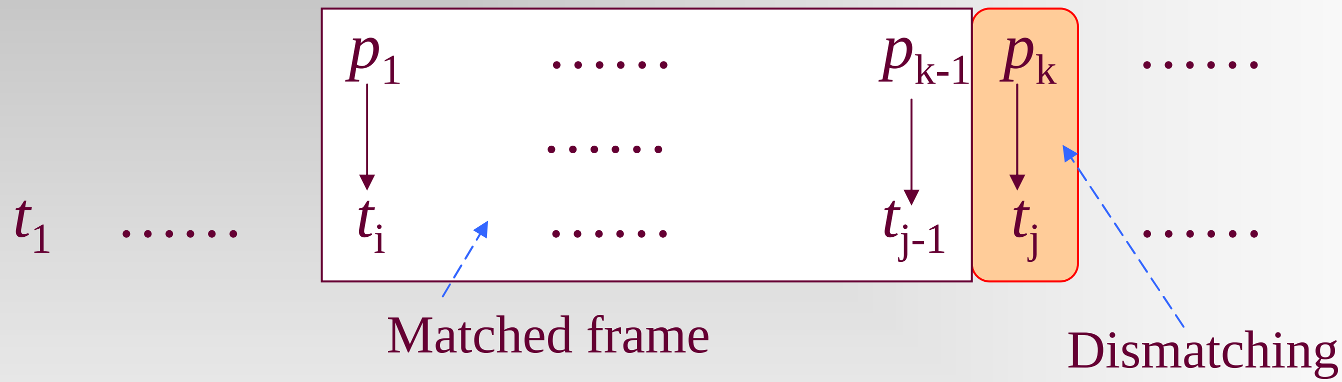
Moving for 4 chars may result in error.

P: ABABABCB

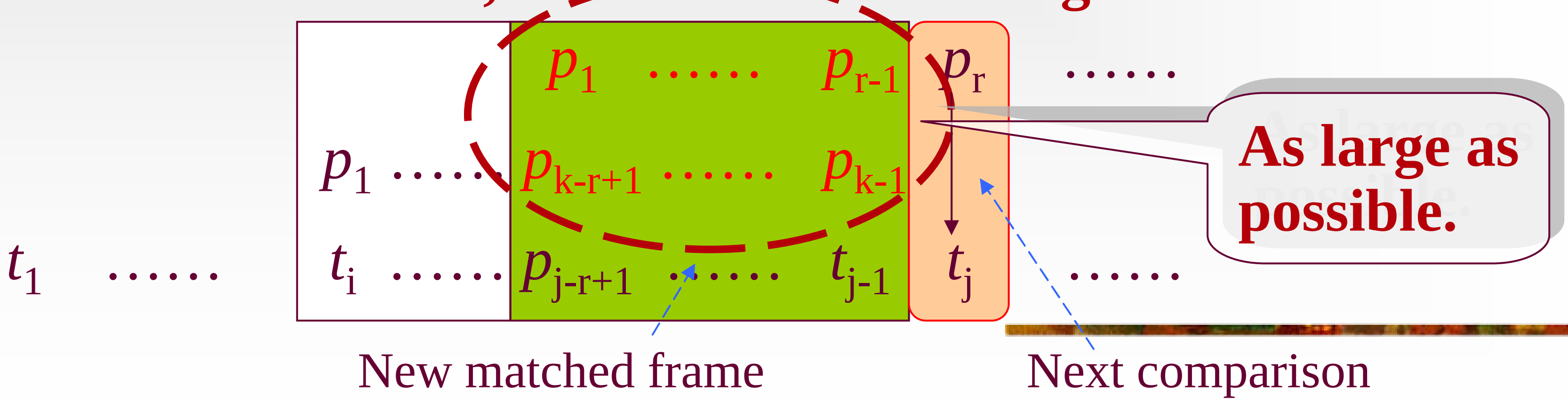
T: … ABABABABCB …

Sliding the Matched Frame

When dismatching occurs:



Matched frame slides, with its breadth changed as well:

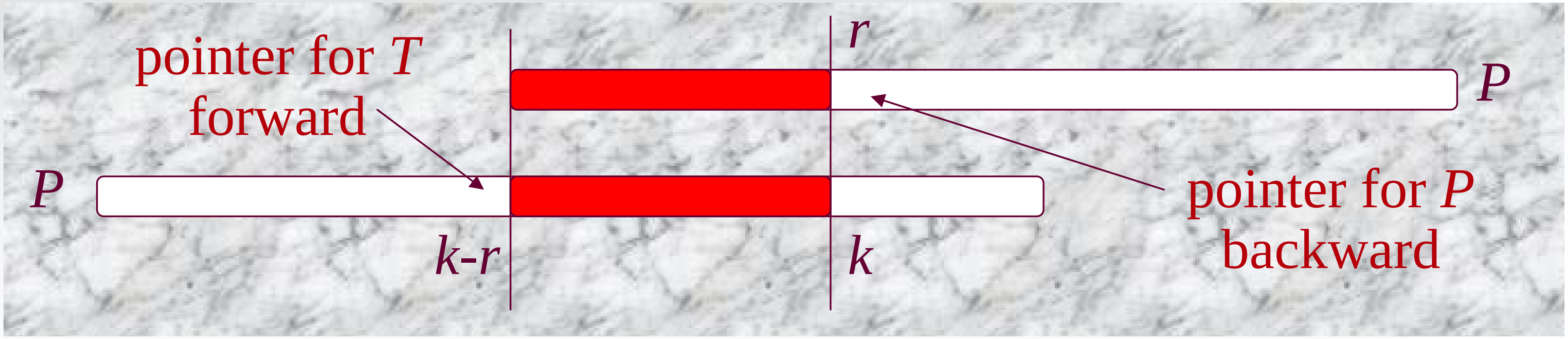


Fail Links

Which means:

When fail at node k, next comparison is p_k vs. p_r

- Out of each node of KMP flowchart is a fail link, leading to node r, where r is the largest non-negative interger satisfying $r<k$ and $p_1,\ldots,p_{r-1}$ matches $p_{k-r+1},\ldots,p_{k-1}$. (stored in fail[k])



- Note: r is independent of T.

Computing the Fail Links

Thinking recursively, let fail[k-1]=s:

To be compared

Matched

$p_1 \quad \cdots\cdots \quad p_{s-1} \quad p_s \quad p_{s+1} \quad \cdots\cdots$

$\cdots\cdots$

$p_1 \quad \cdots\cdots \quad p_{k-r+1} \quad \cdots\cdots \quad p_{k-2} \quad p_{k-1} \quad p_k \quad \cdots\cdots \quad p_m$

To be compared and thinking recursively

Case 1

$\boldsymbol{p_s = p_{k-1}}$

fail[k]=s+1

Case 2: $\boldsymbol{p_s \neq p_{k-1}}$

$p_1 \cdots \ p_{fail[s]-1} \quad p_{fail[s]}$

$p_1 \quad \cdots\cdots \quad p_{s-1} \quad \boldsymbol{p_s} \quad \boldsymbol{p_{s+1}} \quad \cdots\cdots$

$p_1 \quad \cdots \quad p_{k-r+1} \quad \cdots\cdots \quad p_{k-2} \quad p_{k-1} \quad p_k \quad \cdots\cdots \quad p_m$

Recursion on Node fail[*s*]

Thinking recursively, at the beginning, s=fail[k-1]:

Case 2: $p_s \neq p_{k-1}$

$p_1 \cdots p_{fail[s]-1}$ $p_{fail[s]}$

p_1 …… p_{s-1} $\boldsymbol{p_s}$ $\boldsymbol{p_{s+1}}$ ……

p_1 … p_{k-r+1} …… p_{k-2} p_{k-1} p_k …… p_m

p_s is replaced by $p_{fail[s]}$, that is, new value assumed for *s*

Then, proceeding on new *s*, that is:

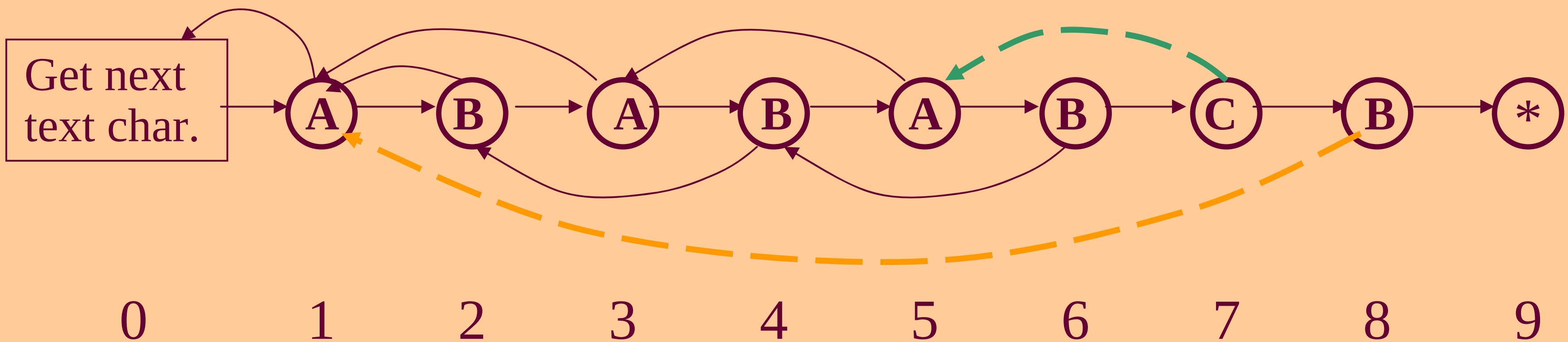
If case 1 applys ($\boldsymbol{p_s = p_{k-1}}$): fail[k]=s+1, or

If case 2 applys ($\boldsymbol{p_s \neq p_{k-1}}$): another new *s*

Computing Fail Links: an Example

Constructing the KMP flowchart for *P* = "ABABABCB"

Assuming that fail[1] to fail[6] has been computed



fail[7]: ∵ fail[6]=4, and $p_6=p_4$, ∴ fail[7]=fail[6]+1=5 (case 1)

fail[8]: fail[7]=5, but $p_7 \neq p_5$, so, let s=fail[5]=3, but $p_7 \neq p_3$, keeping back, let s=fail[3]=1. Still $p_7 \neq p_1$. Further, let s=fail[1]=0, so, fail[8]=0+1=1.(case 2)

Constructing KMP Flowchart

Input: ***P***, a string of characters; ***m***, the length of *P*
Output: **fail**, the array of failure links, filled

```
void kmpSetup (char [] P, int m, int [] fail)
    int k, s;
    fail[1]=0;
    for (k=2; k≤m; k++)
        s=fail[k-1];
        while (s≥1)
            if (p_s = = p_k-1)
                break;
            s=fail[s];
        fail[k]=s+1;
```

For loop executes m-1 times, and while loop executes at most m times since fail[s] is always less than s.

So, the complexity is roughly $O(m^2)$

Number of Character Comparisons

$\le 2m\text{-}3$

```
fail[1]=0;
    for (k=2; k≤m; k++)
        s=fail[k-1];
        while (s≥1)
            if (p_s = = p_{k-1})
                break;
            s=fail[s];
        fail[k]=s+1;
```

These 2 lines combine to increase s by 1, done m-2 times

Success comparison:

at most once for a specified k, totaling at most m-1

Unsuccessful comparison:

Always followed by decreasing of s.

Since: s is initialed as 0,

s increases by one each time

s is never negative

So, the counting of decreasing can not be larger than that of increasing

KMP Scan: the Algorithm

Input: *P* and *T*, the pattern and text; *m*, the length of *P*; *fail*: the array of failure links for *P*.

Output: index in *T* where a copy of *P* begins, or -1 if no match

```
int kmpScan(char[ ] P, char[ ] T, int m, int[ ] fail)
    int match, j,k; //j indexes T, and k indexes P
    match=-1; j=1; k=1;
    while (endText(T,j)=false)
        if (k>m) match=j-m; break;
        if (k= =0) j++; k=1;
        else if ( t_j= =p_k) j++; k++; //one character matched
        else k=fail[k]; //following the failure link
    return match
```

Each time a new cycle begins, $p_1,\ldots p_{k-1}$ matched

Matched entirely

Executed at most 2n times, why?

Home Assignment

- pp.508-
    - 11.4
    - 11.8
    - 11.9
    - 11.13